# BLG460E – Homework I

## Report

### Part I

int get\_uid() {

char buffer[2];

int \*ret;

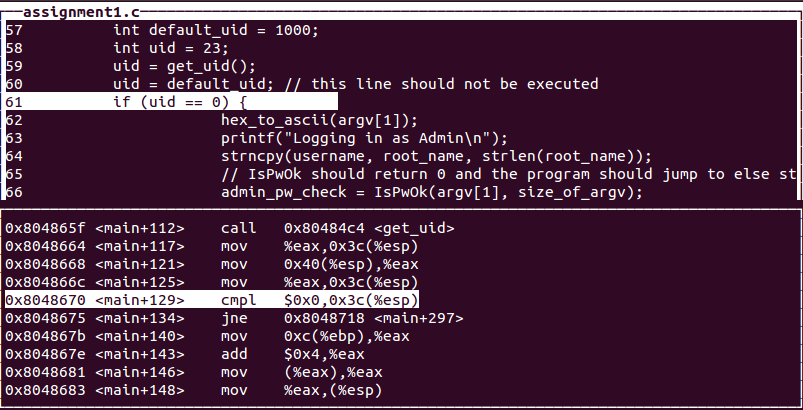
ret = buffer + 10;

(\*ret) += 8;

return 0;

}

The function that I have modified is above. The size of the buffer is (150150150 % 10) + 2 = 2. Firstly, in order to change the return address of this function, I needed to find where it stored in memory.



By using gdb, I have found the return address of the get\_uid() function which is 0x08048664 as it can be seen from the picture above. Thus, in order to skip the if statement and execute else statement, I needed to change this return address.



When I have examined the stack, I have seen that the part of the return address that was needed to change was 10 bytes away from the buffer. That’s why I added 10 to the address of the buffer and assigned it to the ret. Then, since the program needs to go the else statement, I increased the value of return address by 8. Now, the function interferes the program and goes to the address which is required. The stack is shown below after the operations are made.



Finally, the return value of the function is set to 0 in order to assign 0 to the uid.

I have compiled this program by using –fno-stack-protector. If I do not use this command, even though the return address still changes in the get\_uid() function, the program does not skip the next line and executes it. Thus, the program works as if I did not interfere anything.

### Part II

The appropriate input: 32333435f1d1eab73233343582f2ffbfa8f2ffbf02870408

We’ll examine this code byte by byte. The first 8 bytes of this input is for the password. The rest of the input causes the overflow since the size limit of the password is 8. In order to reach the place of the return address, we need to fill the stack with some random values. That’s why the values until the last 8 bytes are garbage values. The last 8 bytes are the return address that the program needs to go.

As I did in the first part, firstly I checked the address of the else statement where the program needs to go after skipping the if statement. As it can be seen the picture below, instead of 0x80486e9, the program should go to the address 0x8048702.

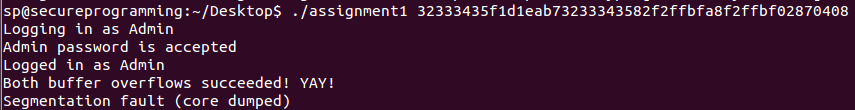


Then, I analyzed the stack and found where the return address of the function is stored. After the finding the position of the return address in the stack, all that needs to be done is to change the address.

Before writing input into stack:

After:

As it can be seen, the first 8 bytes is 32333435f1d1eab7 which is the password that is given. Actually, it has no significance; thus, we filled these spaces with garbage values. Then, there are 12 bytes till the starting byte of the return address. After reaching there, I have changed the return value.



The output of the program is above. Even though I got the correct output, I still got the segmentation fault. When I debug the core code, I discovered that the problem is with the argv[] array. The length of the strings that should be stored in this array should not be longer than 25. However, the length of the input that I gave to the program is 48. Most probably, that causes the segmentation fault.

When we did these operations, we disabled the address space layout randomization. The reason why we did this change is that each time the program runs, it will allocate different places in the memory and that prevents us to change the return address since it will be different each time. However, I tried to run the program by both enabling and disabling ASLR, and the outputs were the same. It did not change anything.